Hardware Implementation of Statecharts for FPGA-based Control in Scientific Facilities

Javier Cereijo García
European Spallation Source
Lund, Sweden
Email: javier.cereijogarcia@esss.se

Roberto R. Osorio
University of A Coruña
Dept. Computer Engineering, CITIC
A Coruña, Spain
Email: roberto.osorio@udc.es

***Abstract*—The problem of generating complex synchronization patterns using automated tools is addressed in this paper. This work was originally motivated by the need of fast and jitter free synchronization in scientific facilities, where a large number of sensors and actuators must be controlled at the right time in a variety of situations. Programmable processors cannot meet the real-time requirements, forcing to use dedicated circuits to produce and transmit the control signals. Designing application specific hardware by hand is a slow and error-prone task. Hence, a set of tools is required that allow specifying the control systems in a clear and efficient way and producing synthesizable HDL (hardware description language) code in an automated manner. Statechart diagrams have been selected as the input method, and this work focuses on how to translate those diagrams into HDL code. We present a tool that analyzes a Statecharts specification and implements the required control systems using FPGAs. A number of solutions are provided to deal with multiple triggering events and concurrent super-states. Also, an alternative microprogrammed implementation is proposed.**

I. Introduction

In this work, we approach the implementation of a software package that allows the automated synthesis of control systems based on Statecharts, a diagram-based tool that widely extends the functionality of finite state machines. In the broader sense, our tool may be used in a variety of contexts. However, it has been motivated and tested to be used in a world-class research center. More specifically, the European Spallation Source (ESS) is a collaboration of 17 European countries to build the world's most powerful neutron source for research [1]. ESS is being built in Lund (Sweden) and will be completed by 2025.

The control system will be based on the Experimental Physics and Industrial Control System (EPICS) [2] software environment, to create distributed soft real-time control systems for scientific instruments. Timing and synchronization (Figure 1) will be based on a global event-based timing system with an event generator at the top of a tree-like structure; event receivers; and fan-out modules. The timing system provides a global distribution of RF-synchronised triggers, beam parameter data and timestamping to the facility, being the event and data frames frequency of 88.0525 MHz.

This work was funded in part by the Ministry of Economy and Competitiveness of Spain, Project TIN2016-75845-P (AEI/FEDER, UE), Xunta de Galicia and FEDER funds of the EU under the Consolidation Program of Competitive Reference Groups (ED431C 2017/04), and under the Centro Singular de Investigación de Galicia accreditation 2016-2019 (ED431G/01).

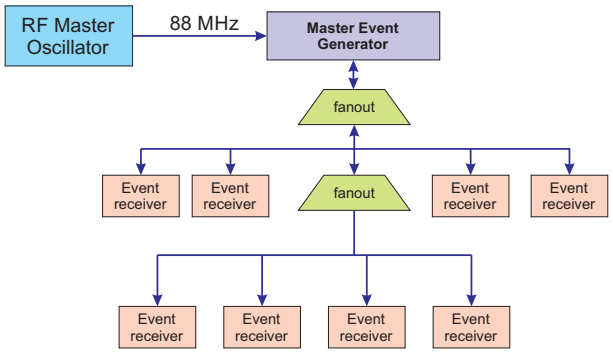

Fig. 1. Timing distribution structure.

For fast signal acquisition and online signal processing, as well as very short latency and hard real time operation, FPGAs are used, particularly, Kintex-7 FPGAs by Xilinx [3]. In each period of the event frequency, an 8-bit event and either an 8-clock distributed bus or an 8-bit packet of the beam parameter data buffer is sent. Figure 2 shows the structure of the event clock cycles in the timing system bit stream. The link frequency is around 1.76 GHz, and the granularity of the timing actions corresponds to one clock cycle, around 11.3 ns. All the changes, triggers, calculations and timestamp resolution should not exceed that time.

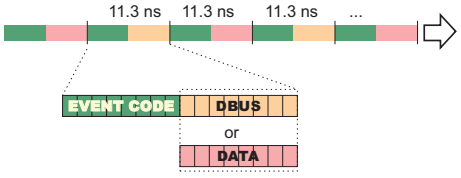

Fig. 2. Frame structure of the timing system bit stream.

The timing system of ESS will have components in different form factors (microTCA, PCIe, VME, standalone, etc) with the same functionality and being compatible with each other, but with different hardware. Because of that, when implementing new functionality it is important to follow a methodology that keeps the chance of errors as low as possible. Hence, a design based on Statecharts complemented with automated synthesis has been considered to implement the aforementioned tasks.

Introduced by Harel [4] in 1987, statechart diagrams allow specifying complex systems in which there may be several states active at the same time and a large number of events and transitions to evaluate. Within the diagrams, basic states may be grouped into super-states and conditions and transitions may be specified at a super-state level. This reduced the complexity of the specification and improved readability.Concurrency is possible, and several super-states may be active at the same time. This makes statecharts suitable to describe complex real-world systems, such as the timing system in a big research facility. The possibility of specifying, without ambiguity, the conditions under which those are enabled or disabled is of great importance. Therefore, Statecharts largely improve traditional state machines, and they become part of the Unified Modeling Language(UML) [5]. Despite they are gaining traction, a reduced number of tools support statecharts. In the context of our current project, an automated way to produce HDL code from a statechart is required in order to deploy a new timing system configuration in a short time without incurring in implementation errors.

The remaining of this paper is structured as follows: in Section II we present an introduction to statecharts and their specification using graphical tools. In Section III, the techniques that allow to synthesize statecharts are described. An alternative implementation based on microprogramming is then proposed in Section IV. Finally, the conclusions of this work are presented.

II. Statecharts

Simple control systems are usually implemented using Finite State Machine (FSM). For more complex systems, however, FSM lack of expressiveness and clarity, as they are simple state-based models that can greatly grow in complexity when adding new states. Also, only one state may be active at any given time, excluding the possibility of implementing concurrency. Finally, outputs and transitions between states may be specified in a state-by-state basis. States cannot be grouped in order to improve readability or specify common behavior.

For those reasons, statecharts were introduced by Harel [4] in 1987. The shortcomings of FSM are solved by allowing for hierarchy, concurrency, and better communication among the states. In this way, compact diagrams are created, on which the concurrency is exposed without ambiguity. States sharing similar outputs and/or transitions may be grouped forming super-states. Communications and transitions may be specified at super-state level, allowing a more clear view of how states and super-states relate to each other. Typically the number of lines and text of a statechart is significantly lower than in an equivalent FSM without sacrificing expressiveness of performance. Hence, statecharts are a visual formalism for describing states and transitions in a modular way. At the same time statecharts maintain all of the characteristics of FSMs, such as conditions, outputs, etc. Their main contributions are:

- Orthogonality: statecharts can have more than one state active concurrently. This goes beyond FSM's capabilities, where only one state can active at a time. Concurrent states are called AND-states, while the traditional approach are called OR-states. Orthogonality is very useful for describing subsystems.
- Depth: the state structure is a hierarchical one, nesting states or super-states inside other states, connected with inter-level transitions. Modularity and clustering is then achieved, improving the design process, and allowing to better specify and understand complex transitions and communications between states at different levels. It is also possible to define entry and default states, and have history in the states, as explained in Section III-C.

An example of a statechart is shown in Figure 3. This statechart consists of 2 super-states at the *top* level named *active* and *wait*. Only one of those super-states may be active at the same time. Hence, *top* is an *OR* super-state. Focusing in, *wait* is also an *OR* super-state on which at a given time either *idle* or *background* may be active, never both. Within *background*, 3 basic states exist. The case of *active* is different, as it is an *AND* super-state made of *send* and *receive*, which are both active or disabled at the same time. Each of them is an *OR* super-state made of several basic states. Hence, this particular statechart may be running either *idle*, or *background* or *send* plus *receive*. Other combinations are not possible.

The transitions between *active* and *wait* are labeled *sleep* and *wake*, and they affect all the internal states without having to specified it on an individual basis. A black dot and an arrow are used to signal the initial node for each super-state. The initial state also applies when a super-state resumes activity, such as switching from *wait* to *active*. Additionally, an H within the dot denotes that 2 super-states have *history*. This means that, when resuming that super-state, the active state will be the one that was running before exiting. In the example, when processing returns to *wait*, the statechart will *remember* whether *idle* or *background* was running and, in the latter case, in which of the 3 nodes.

Statechart specifications may be translated to software or hardware. The former has been already achieved using a variety of tools. The latter was addressed with important limitations in the past [6] [7] [8] [9] [10]. Mainly, history was implemented in a restrictive manner or not at all. Recently, new works have appeared that address the generation of HDL code from statecharts [11] [12], but neither history or transitions between different levels is supported.

A. Graphical tools

Among the tools that allow specifying statechart diagrams using a graphical intergace, we highlight Yakindu Statechart Tools (Yakindu SCT) [13]. There exist many more tools integrated in UML suits, but Yakindu has the advantage of being based on open source tools, such as Eclipse [14], and using an accessible format to store the diagram bases on XML. Yakindu was used to design the diagram in Figure 3, and it allows to specify all the characteristics of statecharts, including transitions between super-states and history.

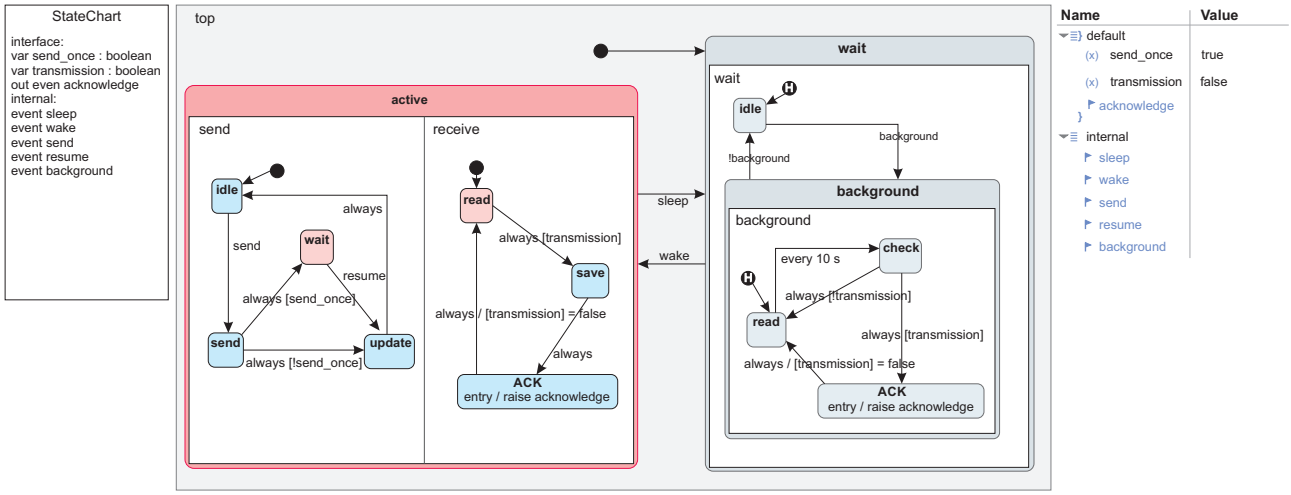

Fig. 3. Example of a statechart in Yakindu SCT.

B. Statechart parsing and analysis

Yakindu SCT saves the Statechart model in a XML file describing the regions, states, history nodes, etc, and the transitions between them. This XML file is organized in a hierarchical manner, with the contents of any given object (element) listed as children. Each element has associated an individual id code in the form of an attribute. Transitions are represented as elements on their own as children of the starting state or history node, and have an attribute pointing to the target element; every element has an attribute with the transition id for all incoming transition. This XML file also describes geometrically the graphical representation of the Statechart model.

The implementation of our application is based on Xerces-C++ [15], a validating XML parser written in C++. Xerces library allows parsing, generating, manipulating and validating XML documents in a variety of ways. In this work, DOM [16] API will be used. Usually, XML documents are navigated in DOM by using a mechanism called tree walker, able to move from node to node in the document. It must be said, that such mechanism did not provide enough flexibility.

Therefore, a more flexible use of the parser was preferred, in which a current node was selected at each step, and other nodes (parent, child or sibling) where accessed by moving back and forth in the structure. This allowed producing HDL code in a well structured manner. Also, the variety of information stored in the XML file requires parsing the document several times, gathering and processing different types of information at each step.

Our application uses the Xerces-C++ parser to produce a synthesizable VHDL file, that implements in an FPGA a statechart modelled in Yakindu SCT. The only input to our application is the XML file describing the statechart, and it detects the inputs and outputs, creates the necessary signals, defines the processes and transitions, and finally generates the VHDL file.

Whereas this work aims to cover most of the features of statecharts, some restrictions have been assumed in the current implementation of the tools. Some of them are parser related, such as ruling out names that include spaces. Some others are implementation oriented, and may require the designer to express the functionalities in a different way. Hence, the statechart will react only to inputs, or events generated within the statechart based on variables or counters. More general events, such as ”wait one second”, cannot be supported. This forces the designer to define a external input or implement a counter that generates an event after one second. It is also required that state transitions happen only between states at the same level of hierarchy and region. Finally, history is implemented, but it will work in a limited way if a deep hierarchy exists. More specifically, only one history state, the more recently used, will be remembered through that hierarchy. This means that some *OR* superstates will resume at its initial state, not their history one.

III. Implementation

The problem of statecharts implementation in software has already being studied and solved [17], mainly in a number of UML tool suits. Translating statecharts to hardware, however, is not solved in a satisfactory manner. Many tools exist of FSM synthesis [18], [19], but most works on hardware synthesis of statecharts have serious limitations, such as not supporting history [8] [12].

Our tool analyses the XML description generated by Yakindu, building a list of basic features such as states, transitions and outputs. From them, concurrency, and hierarchy are extracted; and history is analysed together with the entry and exit conditions.

A. Orthogonal HDL code generation

One of the main differences between FSMs and statecharts is that the latter ones may have more than one state active at the same time. This concurrency is expressed using *AND*

states in the diagram, and must be detected in order to produce concurrent HDL code. It must be taken into account that code may run concurrently a different levels of hierarchy. Usually, *AND* super-states are characterized by producing different outputs, but this is not always the case and it cannot be used as a way to detect concurrency. Even more important, *AND* super-states running concurrently may communicate between them, producing transitions to other states. Using a HDL language, such as VHDL, this behaviour is described using independent processes for each super-state, with a sensitivity list that includes all the inputs, states and conditions that rule the operation of that particular super-state.

In the example in Figure 3, two concurrent processes would be generated for super-states `send` and `receive`, included in `active`. An abridged version of the resulting VHDL code is shown in Listing 1, showing two processes and their sensitivity lists. The operations performed within each processed are not shown, but they basically consist of the evaluation of the current state using a `case-when` construct. On each case, outputs and conditions are produced and the state is updated. Basically, each process implements an individual FSM. Later in this section it will be described how history is implemented, and how to exit and resume the operation in a super-state.

```
sendFSM:process(sleep, wake, send, resume, background,
     sendCurrentState)
begin
  case sendCurrentState is
  [...]
  end case;
end process;

receiveFSM:process(sleep, wake, send, resume, background,
     receiveCurrentState)
begin
  case receiveCurrentState is
  [...]
  end case;
end process;
```

Listing 1. Extract of the output VHDL file implementing the 2 regions in `active` described in Figure 3 (edited for clarification).

B. Hierarchy

Hierarchy is one of the main differentiating factors of statecharts with respect to FSMs. Hence, a mechanism is needed able to implement this characteristic. First, we need to define an strategy to implement how super-states become activated or deactivated during processing. This is achieved by adding a *disabled* state to each super-state. When a super-state looses the focus, the process implementing it will transit to the *disabled* state, and wait there until processing is resumed for that super-state. Resuming consists of a transition to either: the initial state, or the last state if history is implemented.

In a statechart, many transitions are defined at a super-state level, and they affect all children states and super-states. Therefore, the scheme explained in the previous paragraph is incomplete, as it does not consider transitions at a higher level in the hierarchy. In a first implementation, we considered linking each super-state to its parent so that transitions at higher levels would propagate down the hierarchy. However, a more convenient implementation was eventually chosen: all transitions that affect a high level in the hierarchy will be copied to lower levels, making each super-state aware of all the conditions that may affect its operation. In Listing 2, we show an abridged example of the VHDL code produced by our tool that implements super-state `wait` in Figure 3. In it, we can see the *disable* state, called `waitEntry`, that reacts to condition `sleep='1'`. Regarding hierarchy we can see that, despite condition `wake='1'` is specified at a higher level, its evaluation is replicated on children states `idle` and `background`. This means that, despite the specification is hierarchical, the implementation actually flattens that hierarchy.

```
backgroundFSM:process(sleep, wake, send, resume, background
     , backgroundCurrentState)
waitFSM:process(sleep, wake, send, resume, background,
     waitCurrentState)
begin
  case waitCurrentState is
  when idle =>
    if wake = '1' then
      waitHistReg <= 0;
      waitNextState <= waitEntry;
    else
      if background = '1' then
        waitNextState <= background;
      end if;
    end if;
  when background =>
    if wake = '1' then
      waitHistReg <= 1;
      waitNextState <= waitEntry;
    else
      if background = '1' then
        waitNextState <= idle;
      end if;
    end if;
  when waitEntry =>
    if sleep = '1' then
      case waitHistReg is
      when 0 =>
        waitNextState <= idle;
      when 1 =>
        waitNextState <= background;
      end case;
    end if;
  end case;
end process;
```

Listing 2. Extract of the output VHDL file implementing the `wait` region (edited for clarification).

The hierarchy can have any number of levels, so a sub-state in a super-state can at the same time be a super-state. Our application uses recursive functions to go through all the levels of the hierarchy to complete the Statechart model.

C. History

Including a *disabled* state allows exiting and resuming the operation in a super-state. However, implementing history requires that the *disabled* state remembers which state was running prior to exiting (Figure 4.(a)). Two basic strategies can be used to achieve that goal. First, the *disabled* state makes a copy of the current state, which is kept on a register. That value in the register will be used later to resume the operation. This strategy is implemented in Listing 2, and the register is called `waitHistReg`. An alternative option would consist of implementing as many *disabled* states as normal ones. In the case of exiting, the super-state would switch to the *disabled* state associated to the one being processed. For a large number of states, the first option is more desirable. For a small number

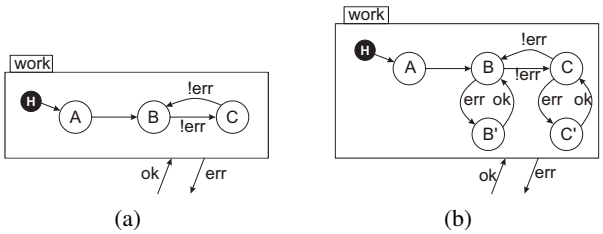

Fig. 4. (a). Superstate with history. (b). Proposed implementation using several *disabled* states

of states, the second option is probably simpler. Thus, the convenience of adopting any of them is case dependent. In the current implementation of our tool, a single *disabled* state and a register are used. However, we show an scheme of how the second option would be implemented in Figure 4.(b).

D. Implementation details

Despite the main aspects of statecharts implementation have been addressed, we have found a number of small challenges during the design of the tool that deserve some explanation:

Most superstates will produce some kind of output or variable updating. Usually, a given output or condition is only modified by a single superstate. Actually, the most used criteria to group states into a super-state is the fact that they produce the same outputs. However, is perfectly possible that an output or condition is modified by several super-states, even in different levels of hierarchy. In that case, an arbitration is needed. Assuming that zero is a neutral value, all super-states producing the same output or updating the same condition may connect to an *or gate*. In most cases, only one super-state will actually produce a valid value, whereas the other ones will produce zero. However, it may not always be that simple, as several super-states may produce a valid output at the same time. In such a case, a proper priority arbiter would be needed.

A clear difference must be made between those events and conditions that are produced and/or evaluated in a single state, or more generally. In the former case, that code should only be part of the specification of that state. In the latter one, a higher level *AND* state must be defined that will be active for longer periods. Possible mismatches between the activity spans of those states that produce events and those that react to them must be studied at design time.

The HDL implementation uses several lists: linking ids and names, linking transitions and triggers, states, and history nodes. For each super-state, signals are defined that account for the current and updated local state, history, and internal variables. Each super-state is implemented as a separate process, in which the local state is updated, conditions are evaluated and outputs are produced. Those processes are purely combinational. An additional clocked process updates states and registered values before starting a new iteration.

IV. Microprogramed implementation

Our tool produces a hardwired control system, which is commonly accepted to be the fastest and more effective implementation. However, an alternative implementation is now proposed, based on microprogramming [20] [21], which was widely used inside microprocessors some decades ago. At that time, computer aided design was not developed, and design errors were quite common. Those errors were often hidden in corner cases and would only show up after the computer was put in the market. Microprogramming allowed vendors to issue control updates and correct those mistakes even in-field.

In some cases, statecharts may be implemented by mapping each super-state into a microprogram. As for microprocessors, the main advantage consists of being able to update the control by just loading a new configuration. There are a number of reasons for using this strategy even on FPGAs:

- configuration changes could be deployed in a short time avoiding logic synthesis, which is a slow task
- updating the microprogram does not depend on the version of the synthesis software, making the control system easier to maintain.

For this purpose, we have implemented a separate tool that allows specifying the number and type of inputs and outputs; a set of internal counters; and the micro-instruction format. Then, it generates the VHDL code that allows implementing the generic control, as well as the circuitry that allows loading new microprograms.

In Figure 5.a we show an example of a microprogrammed control system that is able to implement a variety of super-states. The microprogram is introduced at start up in a word by word fashion. Each configurable register or memory is chained, so that the configuration propagates through all the elements, in a JTAG fashion, until it is fully transmitted. In normal operation, the sequencer analyses the current microinstruction; the inputs; and the events coming from the counters and selects which outputs are produced, as well as the state transitions.

In Figure 5.b, four cases are shown. All of them can be implemented using the same microprogrammed control, by just changing the microprogram. In order to implement the circuit in Figure 5.a, it is necessary to specify all the inputs, outputs and counters that are expected to be used. In the example number 1 in Figure 5.b, there are 4 basic states; and 4 input signals involved. For each state is possible to specify any transition depending on which input is active. In example number 2, one state has been found to be redundant. Removing it does not change the hardware, only the microprogram. A new event is introduced in examples number 3 and 4, signaling when one of the counters reaches a predefined value. The events coming from the counters are treated in the same way as input signals, so only the microprogram has to be changed to support this 2 cases.

Both types of super-states are supported. *OR* super-states are implemented by jumping into different regions of the microprogram. *AND* super-states are supported by implementing 2 or more micro-memories that will run more than one microprogram concurrently. In those cases in which there is not any *AND* super-state active, one or more micro-programs will

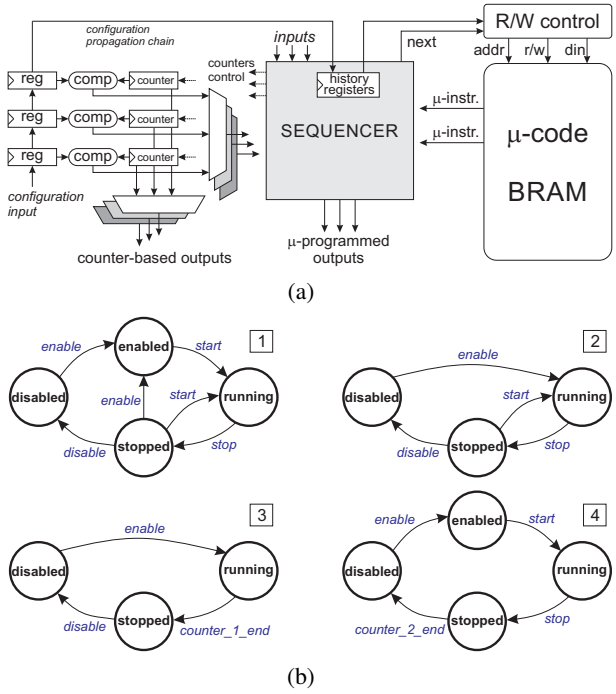

Fig. 5. (a). Microprogrammed implementation. (b). Super-states compatible with the implementation above

jump to an idle state and wait until resuming. This scheme allows dealing even with complex statecharts.

The main limitation to map a super-state into a microprogram is the added delay of accessing the memory plus selecting the appropriate fields within the micro-instruction. Hence, the number of fields in the micro-instructions, which grows with the number of inputs; outputs; and possible transitions, may restrict the applicability of microprogramming. Of course, much research has been done on micro-instruction encoding, so those problems could be tackled to some extent. In any case, a very intricate super-state would require a large number of fields that could make a hardwired implementation more desirable.

It should be made clear that resource consumption is not a problem, even with large microprograms and long micro-instructions, provided that modern FPGAs offer a large number of memory blocks. In the cases shown in Figure 5.b, it can be seen that encoding the transitions would require either 4 fields, 3-bit each; or 6 fields, 2-bit each. Assuming that the possible outputs are the current state and the 2 counters, 4 bits would be required at most.

V. Conclusion

In this paper, we explain the implementation of an automatic tool able to process a statechart diagram and produce synthezisable HDL code. Diagrams may be designed using a graphical tool (Yakindu), which files are parsed and analysed. The procedure to translate the special characteristics of statecharts to VHDL have been explained. Some limitations have been introduced, justifying why those limitations ease implementation. In that sense, our work implements history in hardware in a more general than any other previous paper we are aware of. Orthogonality and hierarchy have been also been implemented without significant restrictions. This work was motivated by the need of implementing complex control systems onto FPGAs in a scientific facility. Our tool will allow to update synchronization policies in a short time without implementation errors. Although our application is already working and produces VHDL code that correctly implements the input Statechart, some updates can be applied in the future. Hence, some of the requirements to the initial statechart can be relaxed, thus allowing for a reduced time creating the statechart. Additionally an alternative design methodology is proposed based on microprogramming, which allows updating the configuration of a control system without depending on logic synthesis. Currently, is not possible to translate a statechart diagram to a microprogram using our tool. However, this will be studied as future work.

References

[1] "European Spallation Source," https://europeanspallationsource.se, accessed: 2019-05-15.

[2] "Experimental Physics and Industrial Control System," https://epics.anl.gov/, accessed: 2019-05-15.

[3] "Xilinx Kintex-7," https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html, accessed: 2019-05-15.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[5] "Unified Modeling Language," http://www.uml.org/, accessed: 2019-05-15.

[6] D. Drusinsky and D. Harel, "Using statecharts for hardware description and synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 798–807, 1989.

[7] P. Clemente, P. Rundstadler, L. Specter, and K. Walsh, "From statecharts to hardware FPGA and ASIC synthesis," in *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, 1992.

[8] V. Salapura, G. Waleczek, and M. Gschwind, "A comparison of vhdl and statecharts-based modeling approaches," in *Proceeding of ITI*, 1994.

[9] T. Muller-Wipperfurth and R. Hagelauer, "Graphical entry of fsmds revisited: putting graphical models on a solid base," in *Proceedings Design, Automation and Test in Europe*, 1998, pp. 931–932.

[10] V. Salapura and V. Hamann, "Implementing fuzzy control systems using vhdl and statecharts," in *EURO-DAC'96, European Design Automation Conference*, 10 1996, pp. 53–58.

[11] S. Qin and W.-N. Chin, "Mapping statecharts to verilog for hardware/-software co-specification," in *FME 2003: Formal Methods*, 2003, pp. 282–300.

[12] V.-A. V. Tran, S. Qin, and W. N. Chin, "An automatic mapping from statecharts to verilog," in *Theoretical Aspects of Computing - ICTAC 2004*, 2005, pp. 187–203.

[13] "Yakindu Statechart Tools," https://www.itemis.com/en/yakindu/state-machine/, accessed: 2019-05-15.

[14] "Eclipse IDE," https://www.eclipse.org/, accessed: 2019-05-15.

[15] "Apache Xerces Project," http://xerces.apache.org/, accessed: 2019-05-15.

[16] "DOM, Document Object Model," https://www.w3.org/TR/dom/, accessed: 2019-05-15.

[17] T. Ziadi, L. Helouet, and J. M. Jezequel, "Revisiting statechart synthesis with an algebraic approach," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 242–251.

[18] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization*. Springer, 1997.

[19] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Logic Optimization*. Springer, 1997.

[20] M. Milkes, "The genesis of microprogramming," *IEEE Annals of the History of Computing*, vol. 8, pp. 116–126, 1986.

[21] *The Design of a Microprocessor*. Springer-Verlag.